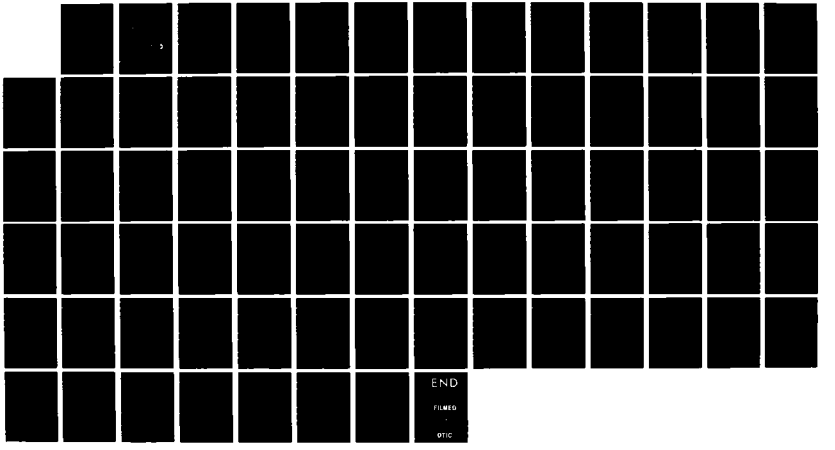
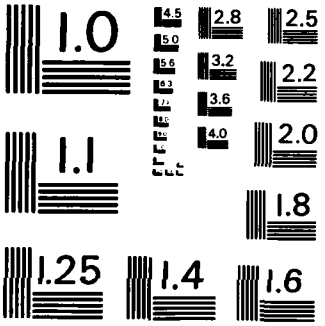
AD-A161 393 DICTIONARY MACHINES FOR CUBE-CLASS NETWORKS(U) ILLINOIS 1/1
UNIV AT URBANA APPLIED COMPUTATION THEORY GROUP
A M SCHWARTZ 05 APR 85 ACT-53 N00014-84-C-0149

UNCLASSIFIED F/G 5/2 NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS - 1963 - A

REPORT ACT-53

MARCH 1985

(12)

DICTIONARY MACHINES FOR CUBE-CLASS NETWORKS

ALAN M. SCHWARTZ

REPORT R-1031

UILU-ENG 85-2206

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION N/A	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release, distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Laboratory, University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Joint Services Electronics Program
6c. ADDRESS (City, State and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code) 800 N. Quincy St. Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER Contract # N00014-84-C-0149
8c. ADDRESS (City, State and ZIP Code) 800 N. Quincy St. Arlington, VA 22217	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: N/A PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Dictionary Machines for Cube-Class Networks	
12. PERSONAL AUTHOR(S) Alan M.Schwartz	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) April 5, 1985 15. PAGE COUNT 68
16. SUPPLEMENTARY NOTATION N/A	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A dictionary is a data structure that supports insertion, deletion, and retrieval operations. To maintain a database, a dictionary machine accepts an arbitrary sequence of instructions at a constant rate. This ~~thesis~~ presents two new VLSI dictionary machines on networks that emulate the binary cube. One machine runs on a shuffle-exchange network. The other machine runs on a cube-connected cycles network. These two designs demonstrate that general purpose networks can perform dictionary operations.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code) 22c. OFFICE SYMBOL NONE

DD FORM 1473, 8' ^PR EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

DICTIONARY MACHINES FOR CUBE-CLASS NETWORKS

Alan M. Schwartz

March 1985

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, U.S. Air Force) under Contract N000-14-84-C-0149.

ABSTRACT

A dictionary is a data structure that supports insertion, deletion, and retrieval operations. To maintain a database, a dictionary machine accepts an arbitrary sequence of instructions at a constant rate. This thesis presents two new VLSI dictionary machines on networks that emulate the binary cube. One machine runs on a shuffle-exchange network. The other machine runs on a cube-connected cycles network. These two designs demonstrate that general purpose networks can perform dictionary operations.

ACKNOWLEDGEMENT

I am forever grateful to the efforts of Professor Michael C. Loui, who served as my thesis advisor. His indefatigable dedication to his students is exceptional.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Overview

This thesis presents new VLSI implementations of dictionary machines. Dictionary machines and priority queues are usually implemented in software, using tries, heaps, and hashing functions. The advent of VLSI enabled faster multiprocessor "hardware" designs to be developed. Several researchers designed systolic dictionary machines based upon tree architectures: the processors being nodes in a binary tree (Leiserson, 1979; Ottmann, Rosenberg & Stockmeyer, 1982; Atallah & Kosaraju, 1985; Somani & Agarwal; 1984). This thesis proposes that the shuffle-exchange network (SEN) (Stone, 1971) and the cube-connected cycles (CCC) (Preparata & Vuilleman, 1981) can also be used to run dictionary algorithms and may offer distinct advantages over tree architectures. By using bitonic merge to insert and delete keys the SEN can easily perform dictionary operations. Likewise, the CCC can also emulate the SEN and is well suited for performing dictionary operations.

The SEN and CCC both enjoy compact VLSI layouts and can be easily programmed. But most importantly, these two structures are very versatile for performing parallel computations. They belong to a family of interconnection networks called the cube-class networks and are able to emulate the binary k-cube and play host to a wide variety of

divide-and-conquer algorithms. These include sorting, calculating the Fast-Fourier Transform, multiplying matrices, evaluating polynomials, performing permutations, and now even performing dictionary operations. Tree architectures cannot match these overall capabilities at the required performance. For example, N numbers can be sorted on a cube and tree machine in $O[\log^2 N]$ and $O[N]$ time, respectively. Two N x N matrices can be multiplied on a cube and tree machine in $O[\log N]$ and $O[N^2]$ time, respectively. Viewed in this light, a cube-class machine is preferred to a tree machine as a general purpose parallel computer, especially if the designer intends to run his dictionary algorithm in conjunction with a wide class of other algorithms (Schwartz, 1980; Synder, 1982).

The format of this thesis is as follows. Chapter 1 includes a definition of the dictionary machine problem and an overview of previous work. Chapters 2 and 3 present the bulk of this thesis: two dictionary algorithms that run on cube-class architectures. The design in Chapter 2 uses bitonic merge to perform dictionary operations on the SEN. It also presents a novel architecture to implement pipelining. This architecture is an important contribution. The dictionary algorithm of Chapter 3 runs on the CCC and borrows heavily from previous work on tree machines. Chapter 4 places these new designs in perspective with previous work and offers concluding remarks.

1.2. Definitions

The task is to maintain a file of records (or pointers to records) dynamically. Records can be inserted, deleted, etc., from the file. Each record is associated with a unique key, and the set of possible keys is totally ordered. The dictionary machine contains the file of key-record pairs (k,r) within its database and performs dictionary operations by searching out the key of each key-record pair. In this paper it can be assumed that any mention of a record also implies its associated key. Operations that return answers are called query operations.

The dictionary operations to be performed are the following:

INSERT(k,r) — add a record to the database.

DELETE(k) — remove the record with key value k from the database.

MEMBER(k) — remove the record with key value k if it is in the database, otherwise answer "not in."

EXTRACTMIN — remove the record with the smallest key from the database.

NEAR(k) — report the record with the smallest key greater than or equal to k.

UPDATE(k) — replace the record with key value k with a newer version.

Ottmann et al. (1982) more precisely define the task. The file F is some ordered set containing key-record pairs (k,r). Let the function F(k) equal (k,r) if k is in F, and null otherwise.

Then:

INSERT(k,r): $F \leftarrow \{F - F(k)\} \cup \{(k,r)\}$.
Response is null.

DELETE(k): $F \leftarrow F - F(k)$.
Response is null.

MEMBER(k): F remains the same.
Response is F(k) if k is in the database.

EXTRACTMIN: $F \leftarrow F - F(kmin)$, where kmin is the smallest key in F.
Response is F(kmin).

NEAR(k): F remains the same.
Response is F(knear), where knear is the smallest key greater than or equal to k.

UPDATE(k): Same as INSERT(k,r) if k is in F, otherwise, no effect.

Some operations are redundant. An insertion is redundant if k already exists in the database. A deletion is redundant if k is not in the database. If a dictionary machine can handle redundant operations then the UPDATE instruction becomes unnecessary (replaced by INSERT).

All processors are initialized with a dummy key-record pair $(\infty,\{\ \})$, where ∞ is taken to be greater than any other key contained in the database. This scheme is used throughout the paper. Also, n refers to the number of keys stored in the dictionary at a particular time, while N refers to the maximum capacity of the dictionary.

1.3. Literature Review

Dictionary operations arise in many applications. A priority queue which uses INSERT and EXTRACTMIN can be helpful in scheduling jobs on an operating system or solving numerical problems iteratively. It also has dozens of other applications (Knuth, 1973). The symbol table, utilizing INSERT, DELETE, and MEMBER, is a practical dictionary. Pattern matching schemes in speech recognition, vision understanding, natural language processing, etc., can be implemented using INSERT, DELETE, and NEAR. Other varied and more complicated uses for the dictionary task are yet to be discovered.

A brief overview of the various dictionary machines is given next.

1.3.1. Uniprocessor designs

All of these are serial algorithms and are well described in textbooks on data structures.

A heap is a binary tree in which keys are stored at all nodes, and the keys stored at the descendants of node x are larger than the key at

x. Thus, the smallest key is always at the root node and is readily available for EXTRACTMIN operations. The priority queue operations INSERT and EXTRACTMIN take O[log n] time.

A height balanced tree can perform INSERT, EXTRACTMIN, and DELETE. Each tree transformation (single and double rotations) takes O[log n] time to keep the tree height balanced.

Hashing functions can also perform dictionary operations. A hashing function assigns to each key a random storage address. INSERT, DELETE, and MEMBER can be performed in O[1] time provided that no collisions occur. In the worst case it takes time proportional to N.

1.3.2. Multiprocessor Designs

These designs employ systolic architectures (Kung, 1982) to achieve a high degree of concurrency and thereby improve performance over uniprocessor designs. Because operations can be pipelined the instruction period is reduced to O[1], thus speeding up the throughput of instructions over serial designs by a factor of log n. The latency, defined to be the time elapsed between issuing a query (e.g., MEMBER) and receiving a response to that query, is O[log n] in a good design.

Systolic architectures are characterized by a large number of small processors interconnected in a simple and regular pattern, such as a two dimensional array or a tree structure. Data flow in a rhythmic,

globally synchronous fashion through neighboring processors; each processor performs a small, local task on the data it receives. Systolic architectures are best suited for highly concurrent, compute-bound algorithms, those whose computation tasks are much greater than its I/O tasks. As an example, the dictionary task is compute-bound because a single I/O operation (e.g., removing a key) may involve a computation on every key in the database (e.g., rearranging the remaining keys). Problems that are especially suited for systolic algorithms include convolution, Discrete Fourier Transforms, matrix arithmetic, graph algorithms, and data structures.

Special-purpose, high performance VLSI designs benefit from using systolic architectures. The simple and regular designs, absence of global communication, and modularity of the systolic designs produce cost-effective VLSI layouts. I/O bottlenecks that might occur with the use of special-purpose devices are reduced considerably because of the high degree of concurrency used. The concurrency is also responsible for the speed-up. All of the dictionary machines described next are based on the principles of systolic architectures and inherit many of the properties just described.

The systolic priority queue (Leiserson, 1979) is a linear array of processors that can sort keys into linear order. Each processor P(i) stores a key, the smallest key being placed in P(1). When a new key is inserted into P(0) it trickles through the array of processors until reaching its sorted position, where it displaces the occupant key. displaced key continues the downward trek looking for its new sorted

position. Since the insertion of new keys can be pipelined, a stream of n keys can be sorted in O[n] time. Such a systolic sorter is used in Chapter 2 as a preprocessor to the dictionary machine and is described in detail there.

Since the minimum value is always found in P(1) and can be retrieved in a single step, the linear systolic sorter can perform EXTRACTMIN in O[1] time. The other query operations, MEMBER and NEAR, require O[n] time to complete. Since all the operations can be pipelined, an efficient priority queue using INSERT and EXTRACT is realizable and has the best performance possible. On the other hand, an efficient dictionary machine is not possible because MEMBER and NEAR have long response times.

In the same paper Leiserson (1979) presents the systolic array-tree which is a binary tree of processors whose leaf nodes are connected into a linear array. See Fig. 1.1. An instruction originates from the root node and is broadcast along the internal tree paths to the processors at the leaf nodes. Every processor receives the same instruction simultaneously and executes it in a single step. An answer to a query is sent up the tree paths back to the root node. The latency is O[log N] and instructions can be pipelined.

Although Leiserson intended the systolic array-tree to be used only as a priority queue, it can perform deletions and membership queries. Redundant operations are not allowed.

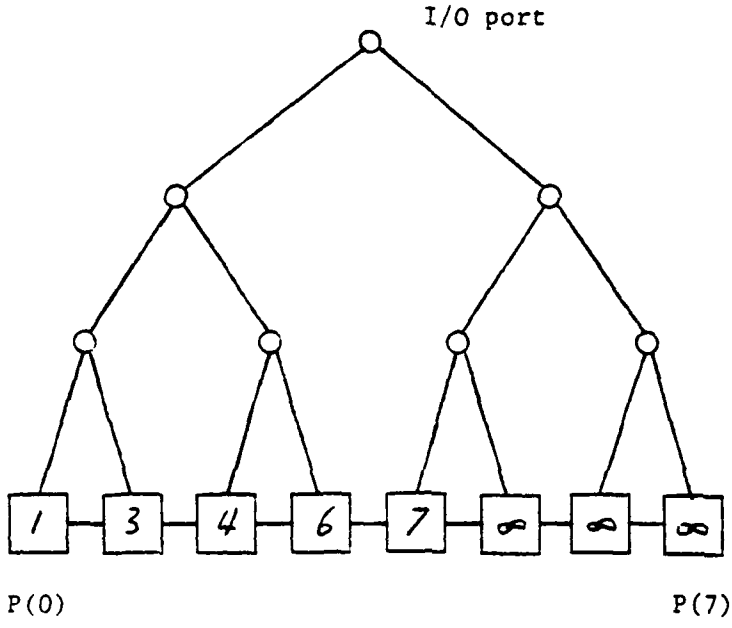


Fig. 1.1. Leiserson systolic tree-array. The keys in the database are (1,3,4,6,7).

An X-tree is a binary tree that has extra wires connecting the nodes across the breadth of a tree level. See Fig. 1.2. It can support all the dictionary operations in O[log n] time, including redundant forms (Ottmann, Rosenberg & Stockmeyer, 1982). The two major improvements over Leiserson's array-tree are the following:

(i) Keys are stored in sorted order along a snake-like chain of processors from root to leaf, called the data path. The net result is to reduce the latency from O[log N] to O[log n] since the largest key is stored at depth log n. Space requirements are also reduced.

(ii) Redundant operations can be handled by allowing holes in the database. To illustrate this concept, consider the following example of deleting an element.

In Leiserson's scheme an element is deleted by shifting the contents of all processors containing a higher key value than the deleted key one position forward (Fig. 1.3(a)). The problem with this scheme is that the wrong key will be deleted if the operation is redundant (Fig. 1.3(b)). Ottmann et al. propose that a hole should replace a deleted key (no shifting occurs), thereby preventing erasure of the wrong key (Fig. 3(c)). Holes are then removed by the use of a COMPRESS instruction (Fig. 3(d)). A COMPRESS instruction is issued once after every insertion and twice after every deletion to ensure that the database will not overflow with an excess of holes.

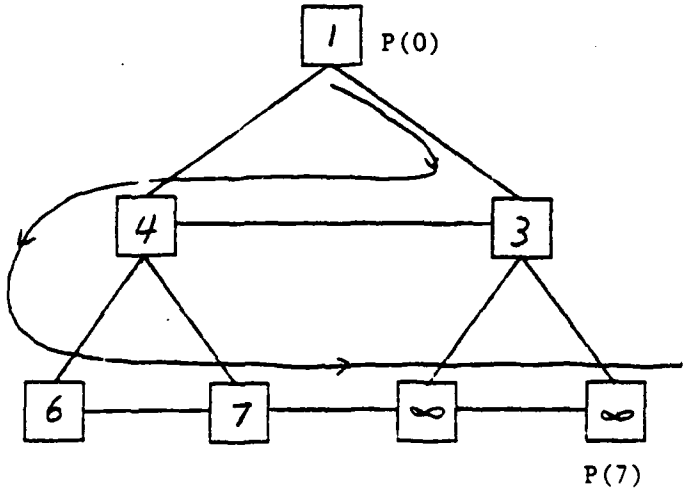


Fig. 1.2. Ottmann et al. X-tree. The keys in the database are (1,3,4,6,7).

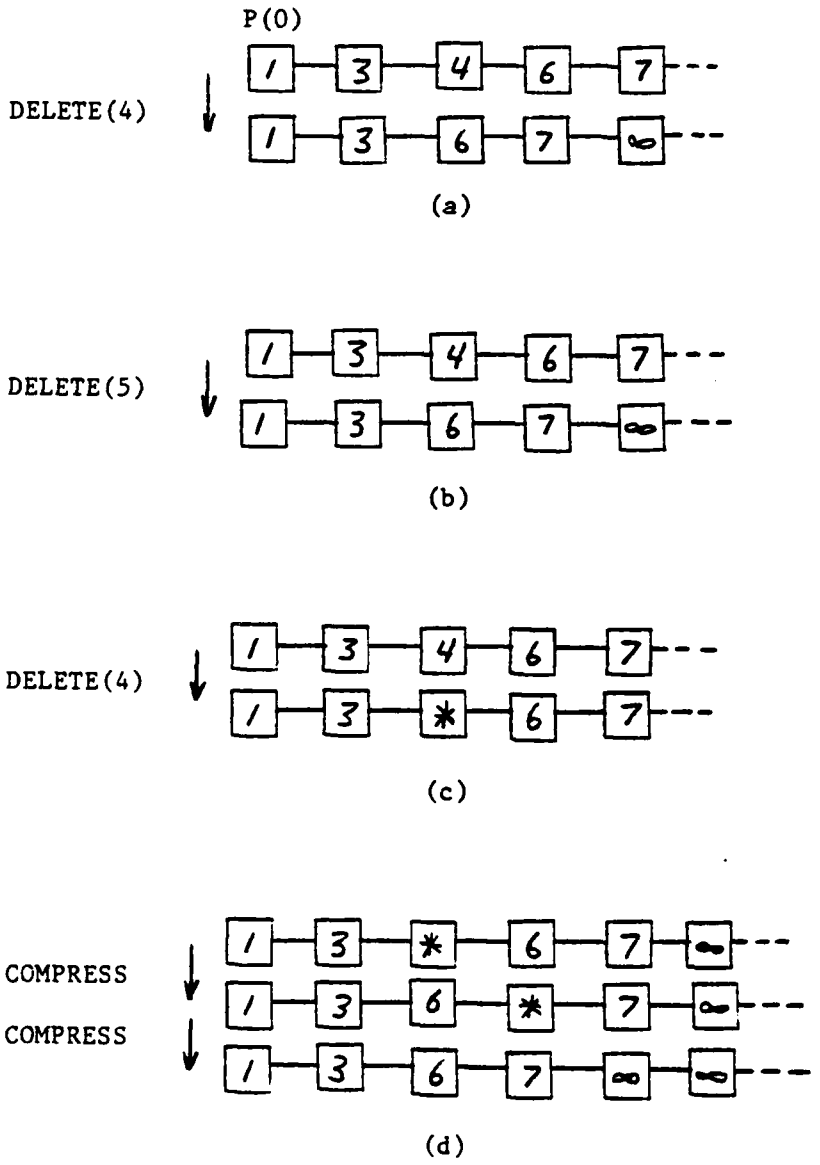


Fig. 1.3. The record in (a) with key 4 is successfully deleted without using holes, but in (b) 4 is mistakenly deleted. Using holes, 4 is deleted in (c), and the hole is removed using two COMPRESS instructions in (d).

The cost of allowing redundancy is a reduction in throughput by a constant factor since a COMPRESS instruction is considered overhead, not a dictionary operation. Also, at any time half the database may be filled with holes so an overhead of N processors is required.

Atallah and Kosaraju (1985) have designed a dictionary machine on a pure binary tree. See Fig. 1.4. The keys are stored in sorted order along the data path in a preordered traversal. This design eliminates the extra horizontal wires found in the X-tree. It enjoys a more efficient VLSI layout than the design of Ottmann et al.

All the dictionary operations have O[log n] latency and can be pipelined. Redundancy is allowed by using the COMPRESS and CLEARTAIL instructions. Also, each processor stores at most three keys.

Unlike the three previous tree machines reviewed here, Somani and Agarwal (1984) have designed a tree machine that stores the keys unordered. The value of a key is not used to determine its placement within the tree. Instead a new key is inserted so that the tree is balanced at each node with respect to the number of keys stored in the left and right subtrees of a node. See Fig. 1.5. To maintain this balance, new elements are inserted alternatively into the left and right subtrees of each node. Since no holes are produced, the overhead of N processors associated with the ordered trees is eliminated.

The housekeeping associated with this design is to rebalance the

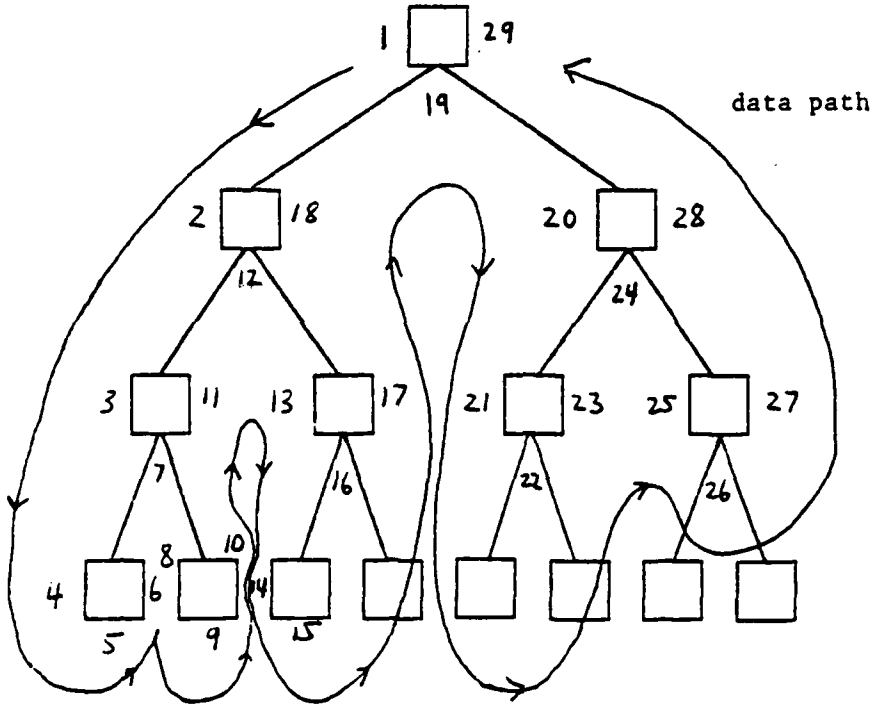


Fig. 1.4. Atallah & Kosaraju's pure binary tree. Three keys are stored at each processor.

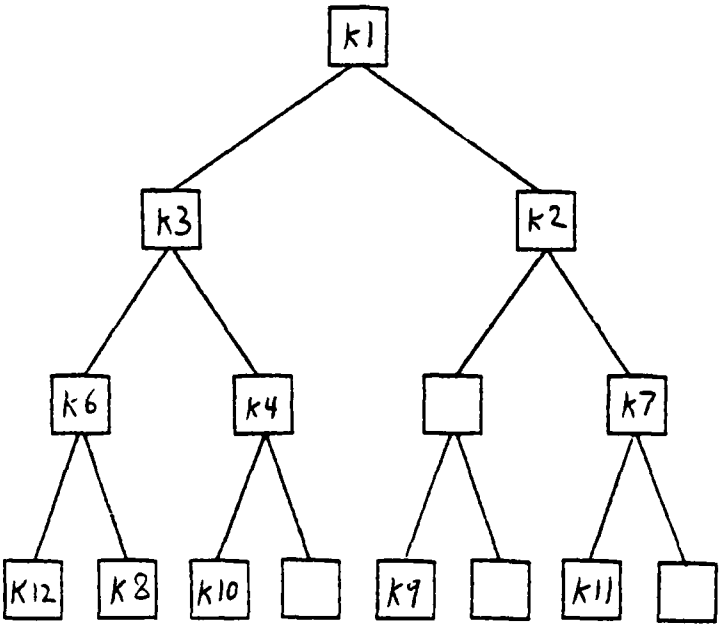


Fig. 1.5. Somani & Agarwal's unordered tree. The i-th key inserted into the machine is denoted by Ki.

tree after a deletion. A single REBALANCE instruction issued after every regular instruction will ensure that the tree is always balanced. Consequently, a constant pipeline interval is maintained.

Carey and Thompson (1984) and Fisher (1984) have designed dictionary machines with O[log N] processors. Each processor P(i) in a linear array is furnished with a local memory. P(i) has twice the amount of memory of its predecessor P(i-1); thus, the largest processor has a memory of size O[N].

In Carey and Thompson's design each processor executes a top-down version of a 2-3-4 tree manipulation algorithm and is responsible for a single tree level. Fisher's design uses a "radix tree" to maintain the database. In both designs NEAR searches are not allowed.

CHAPTER 2

DICTIONARY MACHINE ON A SHUFFLE-EXCHANGE NETWORK

In this chapter we present a dictionary algorithm that runs on a shuffle-exhange network (SEN). Section 2.1 contains a brief review of the SEN and describes the bitonic merge algorithm on which the dictionary algorithm is based. Sections 2.2 and 2.4 show in detail how INSERT, DELETE, and MEMBER are executed on the SEN, and Section 2.3 discusses the novel pipeline method used to achieve a O[1] throughput. Section 2.5 describes the execution of NEAR and EXTRACTMIN. Remarks are presented in Section 2.6.

2.1. The Shuffle-exchange Network

The perfect shuffle is most useful for performing parallel computations (Stone, 1971). A number of parallel algorithms listed in Chapter 1 use the perfect shuffle.

The perfect shuffle maps the addresses i of N elements according to the rule:

$$i \leftarrow 2*i \qquad \text{if} \qquad 0 \leq i \leq N/2 - 1,$$

$$i \leftarrow 2*i + 1 - N \qquad \text{if} \qquad N/2 \leq i \leq N - 1.$$

This is equivalent to cyclically rotating each address one bit position to the left.

Let the SEN have N processors P(0),...,P(N-1), where N is a power of 2. For each i, processor P(i) in the SEN has links to processors:

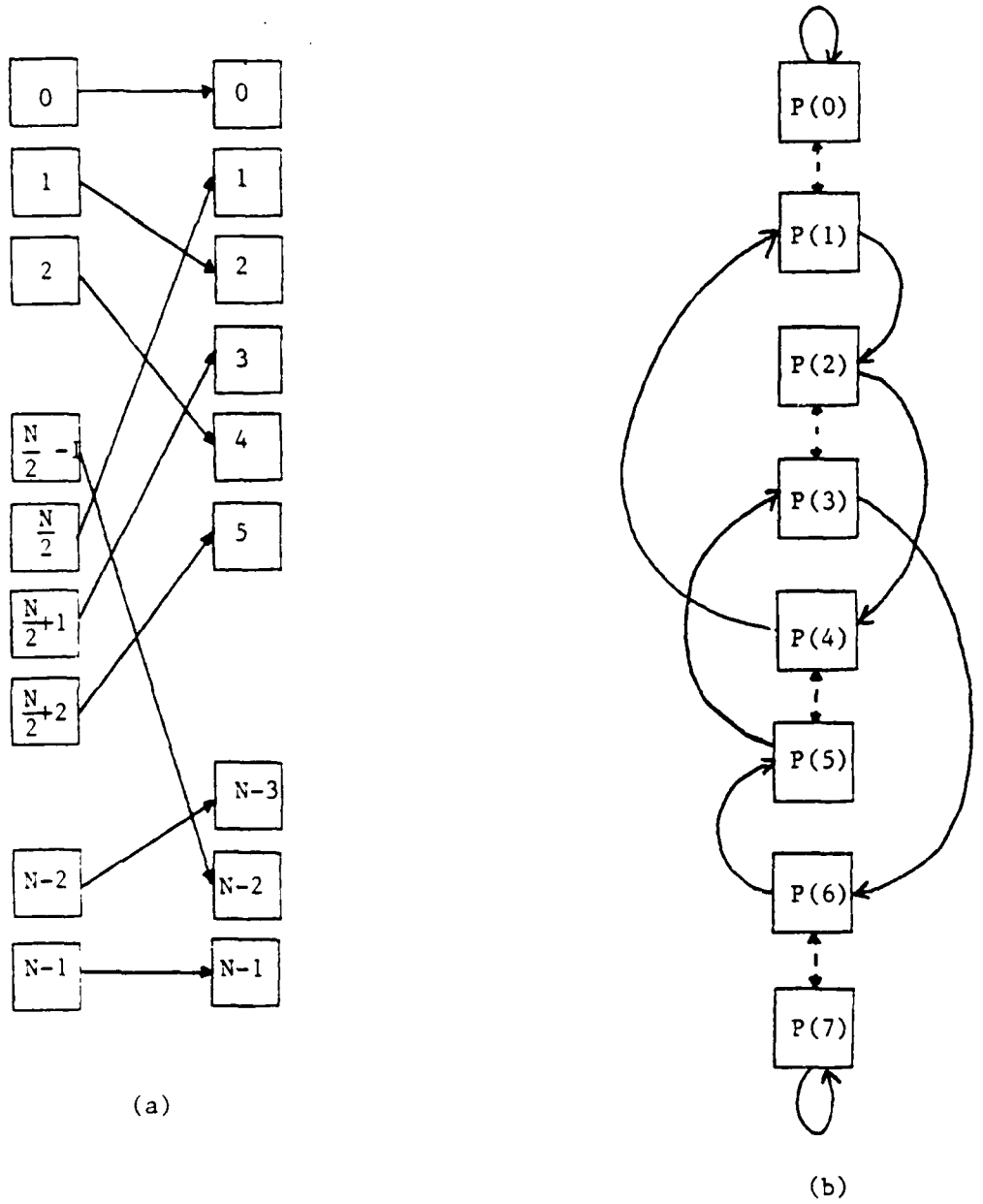
P(i-1) if i is odd (exchange connection),

P(2*i) if $0 \leq i < N/2$ (shuffle connection),

P(2*i +1-N) if $N/2 \leq i < N$ (shuffle connection).

Many divide-and-conquer algorithms can be executed by successively aligning elements that are N/2, N/4, N/8,..., 1 addresses apart and at each step performing a computation on N/2 pairs of data in parallel. The computation executed at each step, for example, could be a comparison-exchange or multiplication, and depends on the algorithm being executed. The ideal network to support these algorithms is the binary k-cube. Each lateral edge of the k-cube that is contained in dimension k is incident on two processors whose addresses differ by a single bit position. Unfortunately, the k-cube is not bounded in degree at each node and so is unrealizable in VLSI.

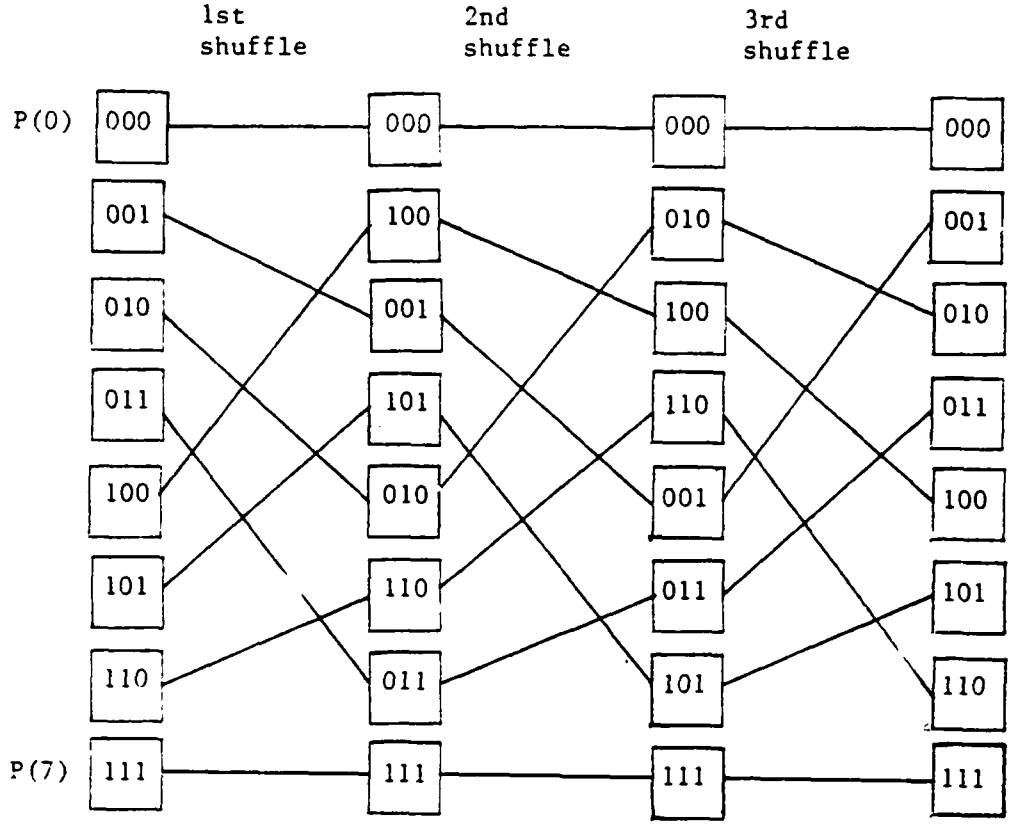
The shuffle-exchange network is an important interconnection network because it can be used as an efficient substitute for the k-cube. It uses the perfect shuffle to align elements whose addresses differ in a single bit position. Figure 2.1(a) shows the perfect shuffle on N elements, and Fig. 2.1(b) shows the SEN on eight elements. Figure 2.2(a) shows three successive shuffles on four pairs of elements and the corresponding addresses of these elements. Figure 2.2(b) shows the same nodes being paired on the k-cube.



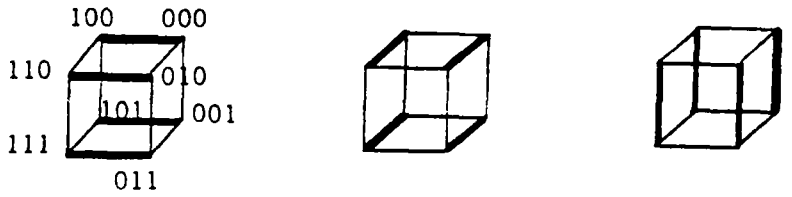
(a)

(b)

Fig. 2.1. (a) The perfect shuffle on N elements and (b) an SEN with eight elements. Broken lines indicate exchange edges.



(a)



(b)

Fig. 2.2. (a) Three successive shuffles of four pairs of elements on the SEN, and (b) their corresponding addresses on the k-cube.

2.1.1. Bitonic Merge on the SEN

A sequence {z1,...,zN} of N numbers is bitonic if $z1 \leqslant,\ldots,\leqslant zK \geqslant,\ldots,\geqslant zN$ for some K, $1 \leqslant K \leqslant N$. Examples of bitonic sequences are {1,2,3,4,5,6,7,8} and {0,2,4,6,5,3,1}.

Batcher's (1968) bitonic merge is a well-known algorithm to merge a bitonic sequence into nondecreasing order. Bitonic merge is a divide-and-conquer algorithm that has an efficient parallel implementation. The bitonic sequence to be sorted is stored in a vector of length N, and the computation to be performed on the sequence is divided among pairs of processors that are successively 2**k-1, 2**k-2,..., 2**0 addresses apart. The computation is a simple comparison-exchange.

Bitonic merge is executed on an N processor shuffle-exchange network in log N steps, where a step consists of (a) a perfect shuffle and (b) a comparison-exchange operation among N/2 pairs of data. The elements are initially stored one to a processor. The bitonic merge algorithm is shown in Fig. 2.3.

2.2. Dictionary Algorithm on the SEN

By storing the records of the dictionary database as a bitonic sequence ordered by key value, all of the dictionary operations can be supported on an SEN machine. The ascending subset of the bitonic

```
Proc BITONIC-MERGE (2^k = N elements)

   for   j←k-1 step -1 until j = 0.
   do    foreach m: 0≤ m <n.
     pardo if bit_j m = 0 then;
                P(m)←max(P(m), P(m + 2^j)),
                P(m + 2^j)←min(P(m), P(m + 2^j)).
                fi
     odpar
   od

corp BITONIC-MERGE
```

Fig. 2.3. The bitonic merge algorithm.

sequence {z1 ,..., zN - log N} represents the N - log N records already present in the database. A single record is stored in each of the first N - log N processors of the SEN. The descending subset of the bitonic sequence {zN - logN + 1 ,..., zN} represents the log N instructions to be executed during the next log N steps. The last log N processors of the SEN constitute the I/O port. Instructions are issued and responses are returned there.

The algorithm supports the operations INSERT, DELETE, MEMBER, EXTRACTMIN, and NEAR. It processes dictionary operations in four distinct cycles: the Execution Cycle, the Merge Cycle, the Response Cycle, and the Compress Cycle, in the order listed. Sections 2.2.1 through 2.2.4 specify the processing of a batch of log N instructions in O[log N] time, provided that the instructions have different key values. In Section 2.4 we show how to modify the Execution Cycle of Section 2.2.2 to process a batch of instructions with identical key values.

In the discussion that follows, an item refers to either a record, a hole, an instruction (defined as the particular instance of a dictionary operation), or a response to a query operation. Every processor contains at each step a response item, and either a record, a hole, or an instruction.

2.2.1. The Merge Cycle

We execute an instruction by placing it into processor N at the

beginning of the Merge Cycle. We also execute a batch of log N instructions by placing them in sorted order into the last log N processors.

The Merge Cycle runs the bitonic merge algorithm in log N parallel steps. At each step the items are shuffled. The item at P(j) proceeds to P(2j) if $0 \leq j < N/2$, to P(2j + 1 - N) if $N/2 \leq j < N$. Then for each even i in parallel, i = 0 ,..., N-2, the items at P(i) and P(i+1) are compared; the item with the smaller key is placed in P(i), and the item with the larger key is placed in P(i+1). Items with the same key value are ordered as follows:

record (k,r) < instruction with key value k < H(k).

In this manner an instruction with key value k will be adjacent to (k,r) if (k,r) is in the database. This last condition is needed to properly execute the instructions.

2.2.2. The Execution Cycle

After the instructions are merged into the database--with the restriction that each instruction in a batch operates on a unique key--instructions are executed in one parallel step. A processor P(i) may use the contents of its two neighbors, (ki-1,ri-1) and (ki+1,ri+1), to determine its new contents.

Because processor P(i) must know the contents of processors P(i-1) and P(i+1), extra links are added between P(i) and P(i+1) for all odd i to facilitate the communication in the SEN. These extra edges are called shift edges. Kleitman, Leighton, Lepley, and Miller (1983) show that shift edges do not significantly increase the layout area in VLSI.

The execution of individual instructions is as follows:

(a) HOLE(k): abbr., H(k). Holes are never issued as instructions outright but are produced during the Execution Cycle as a byproduct of the processed instructions. A hole must retain the key value of the instruction that it replaced since it is considered part of the bitonic sequence. Holes are then removed during the Compress Cycle.

(b) INSERT(k,r): abbr., I(k).

A processor P(i) that contains I(k,r) does:

If ki-1 = k then P(i)←H(k) (redundant insertion),

else P(i)←(k,r).

(c) DELETE(k): abbr., D(k).

A processor P(i) that contains D(k) does:

P(i)←H(k).

Furthermore, every processor in the SEN does:

If P(i+1) = D(k) and ki = k then P(i)←H(k),

(d) MEMBER(k): abbr., M(k).

A processor P(i) that contains M(k) does:

If ki-1 = k then P(i)←MP(ki-1,ri-1)/H(k),

else P(i)←MN(k)/H(k).

Member is a query operation and returns a response item: either MEMBER-POSITIVE(k,r), abbr. MP(k), or MEMBER-NEGATIVE(k), abbr. MN(k). For example, P(i)←MN(k)/H(k) indicates that P(i) contains both a response item and a hole. Response items are returned during the Response Cycle.

The operations NEAR and EXTRACTMIN are discussed in Section 2.5.

We observe that only nonredundant insertions are replaced by new entries into the database. All other instructions are replaced by holes. For example, a DELETE instruction may create two holes, one to replace the deleted key and one to replace the DELETE instruction itself.

The ordering of items established during the Merge Cycle ensures that an instruction with key value k will be adjacent to a record (k,r), if such a record exists. As an example, the sequence {1, D(1), H(1), H(1), 2,...} is possible. The sequence {H(1), D(1), 1, 2,...} is not possible. This ordering ensures that instructions will be properly executed.

Figure 2.4 shows the Merge and Execution Cycles for a batch of instructions.

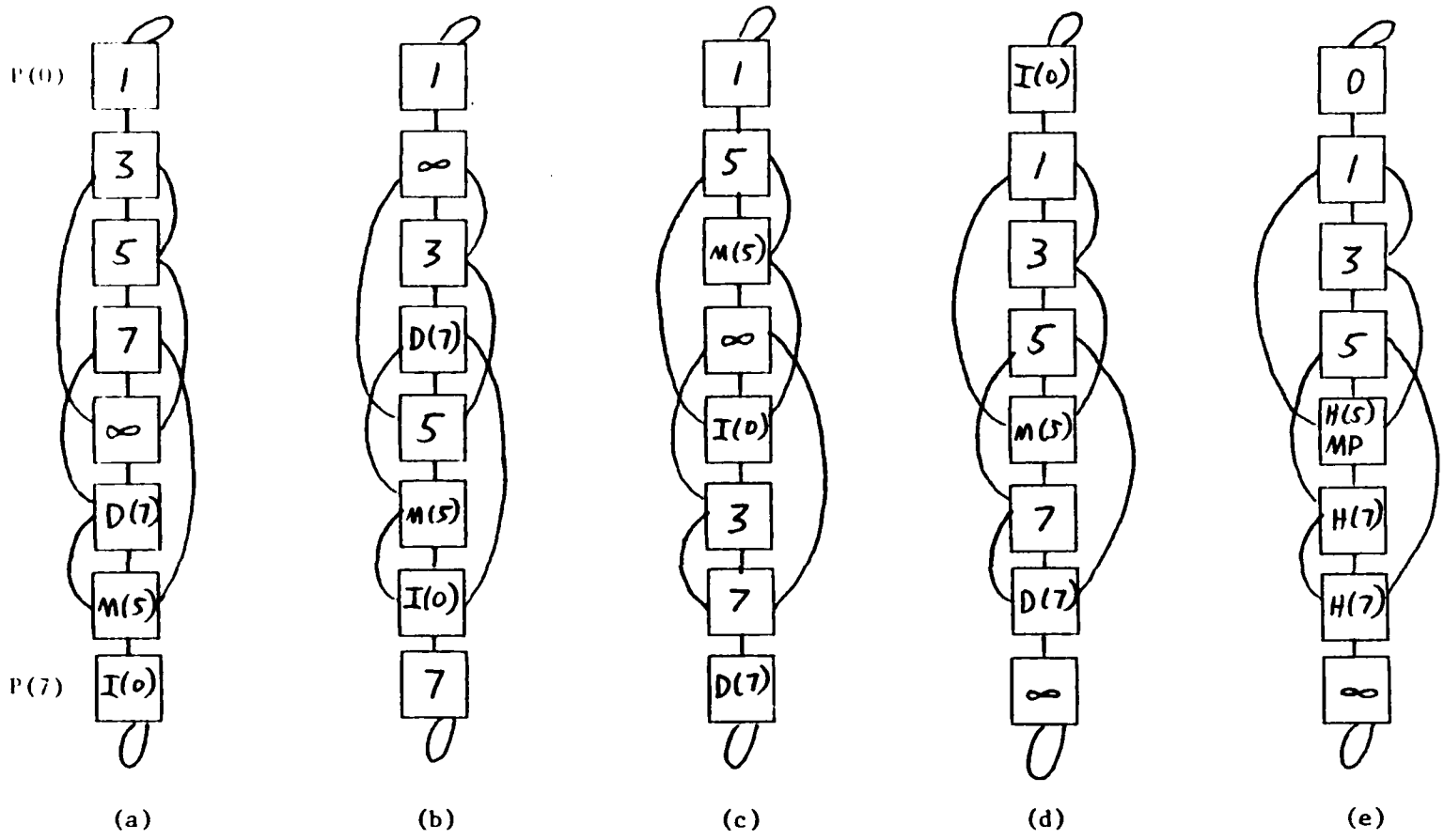


Fig. 2.4. Processing instructions. (a) INSERT, DELETE, and MEMBER in the I/O port of the SEN. (b)-(d) The Merge Cycle. (e) The Execution Cycle.

2.2.3. The Response Cycle

At the end of the Execution Cycle, the database contains records, holes, and response items. During the Response Cycle, response items are returned to the I/O port in sorted key order in log N steps by applying the bitonic merge algorithm backwards. The response items are sent backwards through the bidirectional shuffle links (this is equivalent to the perfect unshuffle) and backtrack the paths they had traced during the Merge Cycle, which will return them to the I/O port. The response items are exchanged when appropriate to 'unravel' the exchanges that had been made during the Merge Cycle.

To exchange response items correctly is a non-trivial task, though. We cannot determine after the fact whether an exhange had been made beforehand unless we record the event when it occurs. Thus, we store the exchange information of each response item in a trace vector. The trace vector is generated during the Merge Cycle and stores the information necessary for the Response Cycle to make appropriate exchanges.

Every response item contains a trace vector. The trace vector is a log N bit register initially set to zero. At each step i during the Merge Cycle, bit i of the trace vector is set to "1" if the item is to be exchanged on step i. Then at each step j during the Response Cycle, a processor pair will exchange response items if bit (log N) - j of either trace vector is set to "1." Otherwise it does nothing.

The records and holes in the database retain their positions throughout the procedure and are unaffected by any data movement in the Response Cycle.

2.2.4. The Compress Cycle

The Compress Cycle takes 4*log N steps. At the outset of the Compress Cycle the database contains records and holes. Roughly speaking, at any time, at most half the items may be holes. We adapt the procedure of Ottmann et al. (1982) to remove holes by shifting them towards the back (i.e., towards processor N - 1) of the machine using a COMPRESS instruction. The COMPRESS instruction is executed simultaneously by every processor in the machine. The algorithm for a COMPRESS instruction is shown in Fig. 2.5(a), and an example of its operation is shown in Fig. 2.5(b).

Following Ottmann et al. (1982), it is known that a COMPRESS instruction must be issued twice for each hole produced to guarantee that the following two conditions hold at the end of the Compress Cycle:

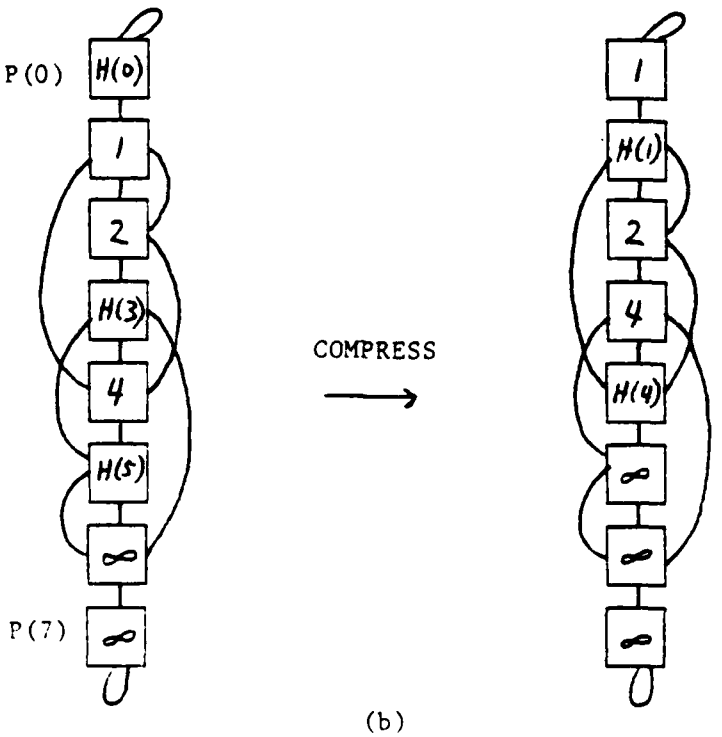
(1) No more than half the processors contain holes.

(2) Neighboring processors do not both contain holes.

The first condition is the basis of an efficient hole removal scheme. It ensures that the machine will not overflow; that is, a record will not be stored into one of the last log N processors. In a

```
proc COMPRESS
    pardo
      If P(i) = H(Ki) and P(i+1) = Ki+1
        then P(i)  Ki+1.
      If P(i) = K =     and P(i-1) = H(Ki-1)
        then P(i)  H(K).
    odpar
corp COMPRESS
```

(a)



(b)

Fig. 2.5. (a) The algorithm for a COMPRESS instruction and (b) its execution on the SEN.

batch of log N instructions, at worst, 2*log N holes are produced if all the instructions are nonredundant deletions. In this worst-case scenario, 4*log N COMPRESS instructions are needed to guarantee that the first condition holds. It follows that we issue 4*log N consecutive COMPRESS instructions during the Compress Cycle. No other processing is done.

As a consequence of the first condition, an overhead of N processors is needed in a dictionary machine that stores up to N records.

The distance between two records in the database is defined to be the absolute difference of their addresses. The second condition restricts the maximum distance to 2 between adjacent records. We use this condition to provide an efficient NEAR search in Section 2.5.

2.2.4.1. Bitonic Sorting

An alternative method for hole removal is to sort holes out of the back of the machine in a single cycle. A shuffle-exchange machine can sort N elements by recursively applying bitonic merge to successively larger bitonic sequences. The bitonic sort algorithm takes $\log^2 N$ steps and is described in Stone (1971). Whenever the machine overflows, in $\log^2 N$ steps, holes can be removed by treating them as infinite key values and applying bitonic sort.

Although this method is asymptotically efficient compared to using COMPRESS instructions, it has two important drawbacks:

(1) The log N pipeline must be periodically stopped to make room for the $\log^2 N$ steps necessary to do bitonic sort.

(2) The distance between adjacent records may be larger than two. An efficient NEAR search may not be possible.

For these reasons, this alternative method for hole removal is recommended only when (1) the NEAR and EXTRACTMIN operations are not being used and (2) it is not necessary to maintain a steady pipeline interval.

2.3. Input/Output Processing

In this section we describe the I/O subsystem which is used to pipeline instructions through the SEN machine. The I/O subsystem acts like a facade on a building; it gives the appearance and behavior of a pipelined architecture, when in fact the internal behavior of the SEN machine is not.

The I/O subsystem consists of a front-end and back-end component. See Fig. 2.6. The front end accepts dictionary instructions serially, at a constant rate independent of N. It processes these instructions into a sorted batch of log N instructions ordered by key value. The batch is then issued into the dictionary machine. The back end takes a

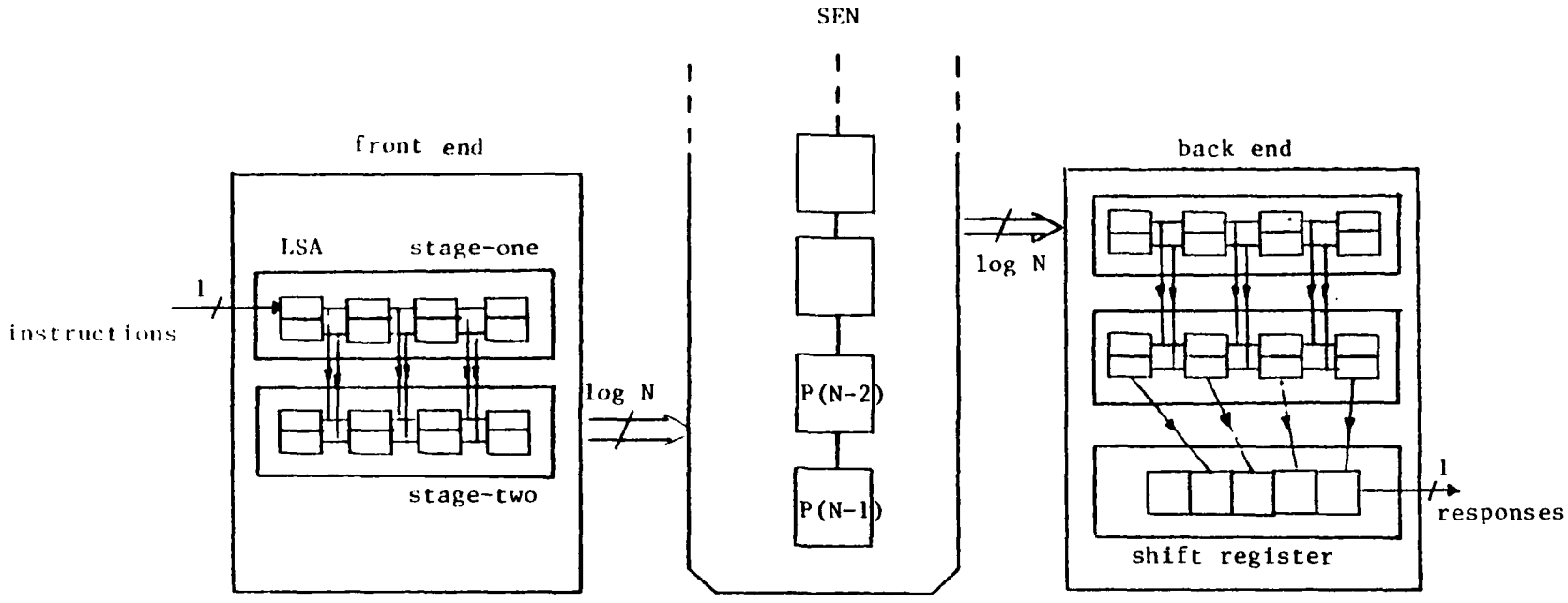


Fig. 2.6. The SEN dictionary machine with I/O subsystem.

batch of response items from the dictionary machine and outputs them serially, in chronological order.

Both the front end and back end are built with Linear Systolic Arrays (LSA's) to effectively process instructions. The LSA has previously been used by Leiserson (1979) to execute the priority queue operations INSERT and EXTRACTMIN. In our application, the LSA's will be used to sort in linear time.

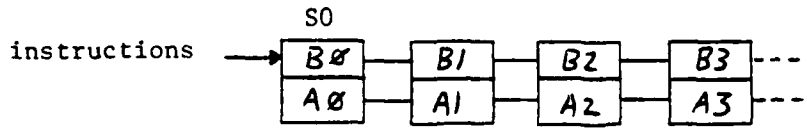
An LSA consists of (log N) + 1 processors, S0,...,Slog N; each processor Si holds two registers, Ai and Bi, and can communicate with its two neighbors Si-1 and Si+1. An LSA is shown in Fig. 2.7(a).

2.3.1. Input Processing

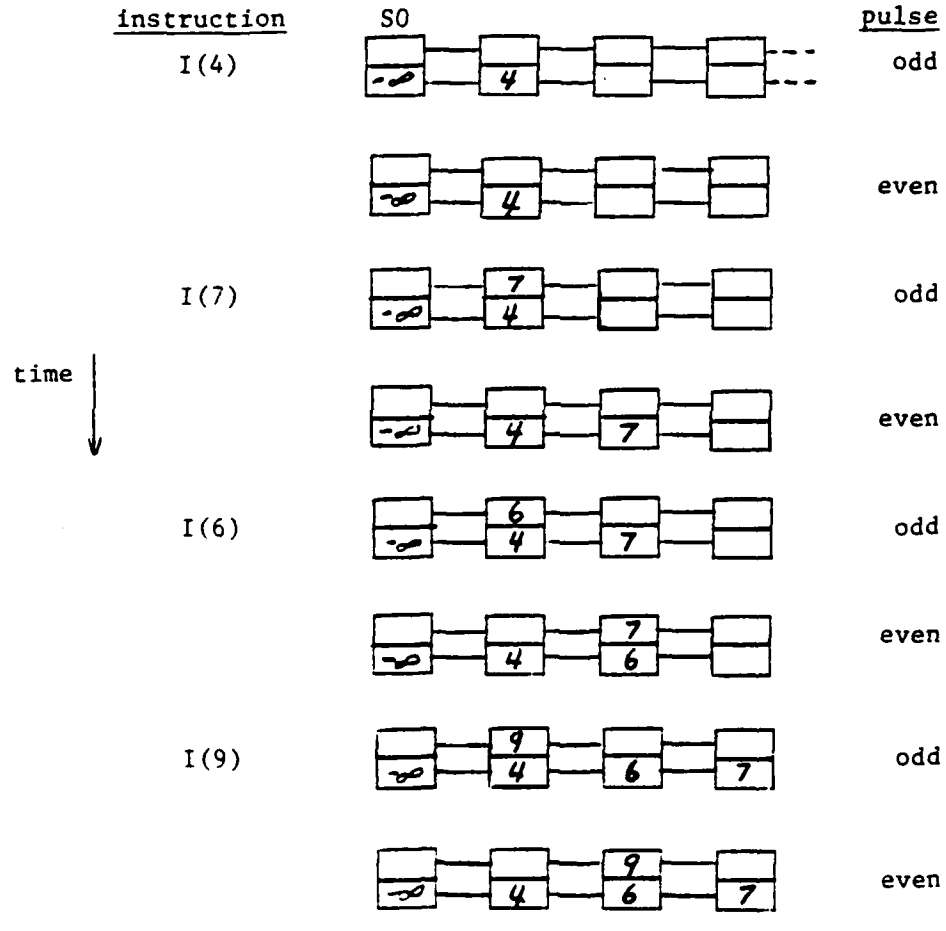
The front end uses a pair of LSA's to process new instructions into a sorted batch of log N instructions ordered by key value. Each LSA can sort in log N steps. Each step comprises an even and an odd pulse.

An even (odd) numbered processor Si executes on the even (odd) pulse the following:

1. Bi←Bi-1
2. Arrange the instructions in Ai-1, Ai, and Bi so that $K(Ai-1) \leq K(Ai) \leq K(Bi)$, where K(Ai) denotes the key value of the instruction contained in register Ai.



(a)



(b)

Fig. 2.7. (a) A linear systolic array. (b) Sorting instructions on an LSA.

Register A0 always contains $-\infty$, and all other registers are initialized with $+\infty$. New instructions are placed into register B0. An example of this procedure is shown in Fig. 2.7(b).

The front end is built from a pair of LSA's configured as a two-stage pipeline. Stage one accepts new instructions and sorts them to the point that a batch of log N instructions has been entered into the LSA. This takes log N steps. It then empties, in parallel, its contents into stage two and immediately begins accepting new instructions.

Stage two finishes the sorting process begun in stage one. This also takes log N steps. It then empties, in parallel, the finished batch of instructions into the input port of the SEN for execution. This step is synchronized with the SEN to coincide with the start of the Merge Cycle.

The stages are connected in parallel; that is, a register Ai (Bi) of stage one is connected to register Ai (Bi) of stage two. In a single step, instructions can be shifted from stage one to stage two. The A registers of stage two are also connected to the I/O port of the SEN. Register B0 of stage two is always filled with + since it does not accept new instructions.

2.3.2. Output Processing

At the end of the Response Cycle, responses to membership queries reside in the I/O port and are ordered by key value. The task of the back end is to sort the responses into their chronological order and output them serially.

The chronological order is determined by a timestamp attached to the items in each processor. The timestamp is a log N bit word, TS, that indicates the issue time of an instruction in a batch. Every instruction is tagged with a timestamp when it enters the front end.

Tagging is done by processor S0 in the following manner:

> On step i, bit i of TS is set to "1."
> All other bits are set to zero.

The timestamp is later used by the back end to rearrange responses into chronological order. The back end must also insert dummy responses to fill in gaps left by regular instructions that return no answers (e.g., INSERT, DELETE).

The back end uses a pair of LSA's to process response items into a sorted batch ordered by timestamp value. Each LSA can sort in log N steps. Processor Si executes at each step the following:

1. Bi←Bi-1
2. If $bit_i TS = 1$ then Ai←Bi,

   where $Bit_i TS$ refers to bit i of the timestamp.

Initially registers Ai and Bi are empty.

The LSA's are configured as a two-stage pipeline and their operation is analogous to that in Section 2.3.1. A parallel load shift register is also used as an output buffer to the pipeline.

2.4. The Modified Execution Cycle

In the Execution Cycle of Section 2.2.2 we assumed that each instruction in a batch of log N instructions operated on a unique key. Here we modify the Execution Cycle to process instructions with the same key value. Write I(k,t), D(k,t), and M(k,t) for INSERT, DELETE, and MEMBER instructions with key value k and timestamp value t. Note that in a batch of log N instructions every instruction has a unique timestamp. Define the timestamp of every record in the dictionary to be zero, and the timestamp of every hole to be $+\infty$. We write R(k,0) to denote the record with key value k, and $H(k,+\infty)$ to denote a hole.

First, in the comparison-exchange step of the Merge Cycle, for all $X,Y \in \{I,D,M,R,H\}$, define $X(k,t) < Y(k',t')$ if either $k < k'$, or $k=k'$ and $t < t'$.

This order relation ensures that at the end of the Merge Cycle, instructions with key value k, denoted by Instr(k,t), appear in contiguous processors between R(k,0) and H(k,+∞) (if both exist). The instructions are ordered by timestamp value and are in chronological order. For example, {R(k,0), Instr(k,t1),...Instr(k,t2), H(k, ∞),...} is a possible sequence of items in the dictionary at the outset of the Execution Cycle.

Next we modify the Execution Cycle to handle this sequence of instructions. The modified Execution Cycle takes 2*log N steps, divided into two parts.

(i) Suppose processors P(q), P(q+1),...,P(q+m) all contain nonhole items with the same key value k. For each j = 1,...,m, in sequence, if processor P(q+j-1) contains an instruction, it executes the instruction as follows:

INSERT(k,r,t):

P(q+j-1)$\leftarrow$R(k,t)

DELETE(k,t):

P(q+j-1)$\leftarrow$H(k,t)

MEMBER(k,t):

If kq+j-2 = k and P(q+j-2) = R(k) then

P(q+j-1)$\leftarrow$ MP(k, R(k))/R(k,t),

else P(q+j-1)$\leftarrow$MN(k)/H(k,t).

In this manner instructions are executed in chronological order. Since m < N, this part takes log N steps.

(ii) After (i) is complete, the old records (and/or holes) associated with k are stored in processors P(q) through P(q+m-1). P(q+m) stores the current record (or hole), and, in log N additional steps, sends back to processor P(q), via the intervening processors, the current record (or hole). Processors P(q+1) through P(q+m) then replace their contents with H(k) to ensure that at most one record is associated with each key.

2.5. Processing EXTRACTMIN and NEAR (with MEMBER)

EXTRACTMIN, abbr., EM, and NEAR, abbr., N(k), can be processed between parts (i) and (ii) of the modified execution cycle in 4*log N steps. MEMBER is also processed with NEAR and EXTRACTMIN. Thus, the new order of execution is as follows:

(i)	1st part of INSERT and DELETE	takes 1 step
(ii)	EXTRACTMIN and NEAR	takes 4*log N steps
(iii)	MEMBER	takes log N steps
(iv)	2nd part of INSERT, DELETE, and MEMBER	takes log N steps

2.5.1. EXTRACTMIN

Every EXTRACTMIN instruction has key $-\infty$. EM($-\infty$,t) with key value $-\infty$ and timestamp value t responds with the record R(k',t'), where k' is the smallest key in the database such that t > t'.

First consider a batch of instructions that contains just one EM instruction. EM($-\infty$,t) is merged into the database during the Merge Cycle and resides in processor P(0). Since at the start of the Merge Cycle, neighboring records are separated by at most one hole (see Section 2.2.4), then R(k',t') must reside in a processor P(j), $0 < j \leq 2*\log N$. Thus, in $2*\log N$ steps, EM($-\infty$,t) can search and find R(k',t') by "walking" through the intervening processors between P(0) and P(j). Since EXTRACTMIN is processed between parts (i) and (ii) of Section 2.4, all the "old" records are still available in the database for retrieval (they have not been replaced by holes yet). When EM($-\infty$,t) finds R(k',t'), it deletes it and creates a response item denoted by MIN(R(k'),t).

The response to EXTRACTMIN must be sent back to P(0) since the trace vector of the response is associated with P(0), not P(j). This takes an additional $2*\log N$ steps. The response is then returned to the I/O port during the Response Cycle. Then, on the last step, EM($-\infty$,t) in P(0) is replaced with H(k).

If m EXTRACTMIN instructions are processed in the same batch, they

are merged into the first m processors of the SEN. Associated with the batch is a log N bit register, EMR, that has bit i set to "1" if EM($-\infty$,i) is in the batch. Thus, EMR stores every EXTRACTMIN instruction of the batch. This register initially resides in P(m). The EXTRACTMIN instructions residing in P(0) through P(m-1) are used only to occupy space for the forthcoming response items.

In 2*log N steps, the EMR register then walks through processors P(m+1) to P(m + 2*log N) and when possible, executes any of the EXTRACTMIN instructions it has stored. Execution is as follows:

For j = 1 to 2*log N;

if P(m+j) = R(k,t) and there exists an integer b, where
b is the smallest integer greater than or equal to t such that
bit$_j$EMR = 1,
then:
P(m+j) $\leftarrow$ MIN(R(k),b)/H(k,t),
bit EMR $\leftarrow$ 0.
else, do nothing.

At each step, P(m+j) executes the oldest EXTRACTMIN instruction that is younger than the record R(k,t) stored in P(m+j). If it has one then it creates the response item, deletes R(k,t), and discards the EM instruction from the EMR register.

In 2*log N additional steps, the responses are sent back to the first P(m) processors and then returned to the I/O port during the Response Cycle.

2.5.2. NEAR

N(k,t) responds with the record R(k',t'), where k' is the smallest key such that $k' \geq k$ and $t > t'$. Processing is similar to EXTRACTMIN except that NEAR is merged into the database with key value k to reside in some processor P(i). R(k',t') will reside in some processor P(j), $i \leq j \leq i + 2*\log N$. N(k,t) can search and find R(k',t') in 2*log N steps. When found, N(k,t) creates the response item N(R(k'),t). R(k',t') is not deleted. In 2*log N mdditional steps, the response is sent back to P(i) and then returned to the I/O port during the Response Cycle. If several NEAR instructions are processed in the same batch they may respond with the same record. In this case, the response in P(j) is sent back to several processors, P(i).

Now suppose N(k,t) has responded with a record R that EM($-\infty$,t') ($t' < t$) wants to claim and delete. In this case, NEAR has responded erroneously. To solve this problem, N(k,t) should claim R and then continue searching for several more records. When EM($-\infty$,t) rightfully claims R, N(k,t) will have found another record. Although N(k,t) may claim several records in its search, the first record to respond back to P(i) will be the correct response.

2.5.3. MEMBER

Member is processed after EXTRACTMIN and is identical to the procedure in (i) of Section 2.4. This takes log N steps. Notice that INSERT and DELETE can now be processed in a single step.

2.6. Remarks

The dictionary algorithm takes O[log N] steps. To summarize we have:

Cycle	Processing time
Merge:	log N steps
Response:	log N
Compress:	4*log N
Execution (with INSERT, DELETE & MEMBER):	2*log N
total	8*log N
Execution (and NEAR & EXTRACTMIN):	6*log N + 1
total	12*log N + 1

It takes at most 12*log N + 1 steps to process a batch of log N dictionary instructions on the SEN.

CHAPTER 3

DICTIONARY MACHINE ON A CUBE-CONNECTED CYCLES

In this chapter we present a dictionary algorithm that runs on a cube-connected cycles (CCC). Sections 3.1 and 3.2 describe the dictionary algorithm and its salient characteristics. Section 3.3 contains a brief review of the CCC and establishes two graph properties needed to apply the algorithm. Section 3.4 presents the dictionary algorithm on a CCC. Section 3.5 offers concluding remarks.

3.1. The Dictionary Algorithm

The algorithm maintains a database of ordered key-record pairs (ki,ri), stored one to each processor. The ordering is by key value; that is, $k_{i-1} < k_i < k_{i+1}$, $1 \leq i \leq N$. The smallest key is stored in P(1).

Computation is systolic. At each step all processors execute the same instruction. A processor P(i) may use the old contents of its two neighbors P(i-1) and P(i+1) to determine its new contents. Initially, each processor contains $+\infty$. A hole is denoted by (ki,*). As described by Ottmann et al. (1982) each processor in a single step does:

INSERT(k,r) — If $k_{i-1} < k \leq k_i$ then $(k_i,r_i) \leftarrow (k,r)$.
If $K_{i-1} = k$ then $(k_i\ r_i) \leftarrow (k,*)$.
If $k < k_{i-1}$ then $(k_i,r_i) \leftarrow (k_{i-1},r_{i-1})$.

DELETE(k) — If $k_i = k$ then $(k_i,r_i) \leftarrow (k,*)$.

MEMBER(k) — If $k_i = k$ and $r_i \neq *$ then answer (k_i,r_i);
else answer "not in."

EXTRACTMIN — If $i = 1$ and $r_1 \neq \{\ \}$ then answer (k_1,r_1);
else answer "not in."
$(k_i,r_i) \leftarrow (k_{i+1},r_{i+1})$.

COMPRESS — If $r_i = *$ an $r_{i+1} \neq *$ then $(k_i,r_i) \leftarrow (k_{i+1},r_{i+1})$.
If $r_i \neq *$ and $r_{i-1} = *$ then $(k_i,r_i) \leftarrow (k_i,*)$.

NEAR(k) — If $k_{i-1} < k$ and $k_i > k$ and $r_i \neq *$,
then answer (k_i,r_i);
else answer "not in."

The algorithm handles both redundant and nonredundant instructions by using holes. Every DELETE and redundant INSERT instruction creates a single hole. Holes are removed by shifting them towards the back (i.e., towards processor N-1) of the machine using a COMPRESS instruction. See Section 2.2.4. The COMPRESS instruction should be issued twice after every instruction to ensure that holes are removed efficiently.

At most 2*N processors are needed to store N-1 records in the machine. This overhead can be reduced considerably if extra COMPRESS instructions are executed. If COMPRESS is executed c times after every instruction, then at most [N/(c-1)] extra processors are needed.

3.2. Communication in a Dictionary Machine

In a pipelined machine with many processors, communication must be carefully planned to ensure that data, instructions, and responses flow accurately. Sections 3.2.1 and 3.2.2 present the communication scheme. We show that when a network has both a Hamiltonian path and no odd length circuits, efficient communication is always possible. Section 3.2.3 describes retiming, which uses buffers to solve timing problems associated with pipelining.

3.2.1. The Data Path

This is defined to be the simple path connecting neighboring processors P(0),..., P(i-1), P(i), P(i+1),..., P(N-1) in a network of processors. The network may be, for example, a mesh, a tree, or a CCC. The links along the data path are used to compare and move records during the execution of the dictionary algorithm.

The data path coincides with a Hamiltonian path (HP), thus establishing that every processor has a unique address between 0 and N-1. An HP is a path that visits every processor in the network once.

If the endpoints of the path are also connected, then the path is a Hamiltonian circuit (HC).

3.2.2. Instruction and Response Paths

All the interconnections of the network are used to broadcast instructions. Instructions originate from the root node P(0), and are broadcast along the forward links. Responses are returned along the back links. Every interconnection contains a forward and a back link, and together they form a bidirectional pair. An interconnection may also contain a data link if it connects processors P(i) and P(i+1), $1 \leq i \leq N-2$, along the data path.

A processor receives instructions through its incoming port(s) and forwards instructions through its outgoing ports(s). A forward link (x,y) will connect an outgoing port of P(x) to an incoming port of P(y). P(x) is a predecessor of P(y), and P(y) is a successor of P(x).

We compute the incoming/outgoing ports, and equivalently the forward and back links, during the initialization step. This step issues a message from the root node to all processors in the network. Each processor labels the port(s) that receive the message as incoming. It labels the other ports outgoing. The message is then forwarded through the outgoing port(s). In this manner, every port in the network is labeled, and the communication paths are formed.

The message also contains a counter which is initialized to zero by P(0) and incremented by one each time it is received by a processsor P. The counter message is stored as the distance from the root to P.

The message visits the nodes in an order characterized by a breadth-first search (BFS). Consequently, the forward links will induce a directed acyclic graph (DAG) on the network. A DAG is a directed graph that contains no circuits. In a DAG, the distance from the root to node x is called the layer. □

Lemma 3.1: Let G be a graph with no odd length circuits. In any DAG induced by a BFS of G, every edge is between layers.

Proof: If a directed edge (x,y) is not between layers, then x and y are in the same layer, and it would be possible to trace two separate paths from the root to x and y, respectively. These paths would have the same length and could be connected via the (x,y) edge. This contradicts the assumption that the graph has no odd length circuits.

In a network with no odd length circuits, the initialization step by lemma 3.1 ensures that the forward links are between layers. This establishes that every instruction is received only once by each processor. Furthermore, a processor having two or more predecessors will receive the same instruction from each predecessor.

Likewise, a processor having two or more successors will receive

responses to the same query instruction from each successor. The responses are combined to yield either a record or a "not in" and then forwarded along the back links. In this manner, instructions and responses can be pipelined through the network free of contention or timing problems.

3.2.3. Retiming

If P(i) is a successor of P(i+1), then they receive the same instruction one time step apart. In a tree machine this happens when neighboring processors are at two different tree levels. In a CCC this happens to every processor. To accomodate the dictionary algorithm, which is synchronous and expects P(i) and P(i+1) to store the current record simultaneously, we retime the network (Leiserson & Saxe, 1984) and use buffers.

Each processor P handles an instruction in four steps:

1. Receive and decode instruction.
2. Forward instruction to successors.
3. Save current record (or hole) in a buffer.
4. Execute instruction, possibly changing the record at P.

In this manner, when P is at step 4 and executing the current instruction, its successors are at step 2 of the current instruction, and its predecessors are at step 2 of the next instruction. To execute the current instruction P can use the contents of its successors and the

contents of the buffers of its predecessors. An example of this procedure is shown in the timing diagram of Fig. 3.1.

3.3. The Cube-connected Cycles

The CCC (Preparata & Vuillemin, 1981) is a multidimensional cube in which the cube vertices have been replaced by a cycle of processors. Because the binary k-cube has unbounded degree and is unrealizable in VLSI, the CCC was introduced as an efficient substitute. The CCC can emulate the SEN using a combination of pipelining and parallelism. It also enjoys a simpler VLSI layout than the SEN.

We denote by CCC(N) a CCC with N = 2**K processors. Define r(N) to be the smallest integer r such that r + 2**r $\geq$ K. CCC(N) has 2**(K-r) cycles. Each cycle contains 2**r processors.

There are three types of interconnections within a CCC. Forward and backward edges connect processors within a cycle, and lateral edges connect processors between cycles. Every module contains three I/O ports, one for each type of interconnection.

Processor Y in cycle X is addressed by P[X,Y], where $0 \leq X \leq 2^{**}(K-r) - 1$ and $0 \leq Y \leq 2^{**}r - 1$.

The interconnections are as follows:

1. P[X,Y] connects P[X, (Y + 1)*mod(2**r)].

Processor \ Time	1	2	3	4	5	6	7
P(1)	R1	F1	S0	E1	R2	F2	S2
P(2)			R1	F1	S0	E1	R2
P(3)					R1	F1	S0

Ri: Receive instruction i.
Fi: Forward instruction i.
Si: Save the result of instruction i in a buffer.
Ei: Execute instruction i.

Fig. 3.1. Timing diagram of instruction execution. At time 4, P(1) uses the contents of P(2) to execute instruction 1. At time 6, P(2) uses the contents of the buffer of P(1) to execute instruction 1.

2. P[X,Y] connects P[X, (Y-1)*mod(2**r)].

3. P[X,Y] connects P[X + e*2**Y, Y],
   where e = 1 - 2*bit$_Y$(X).

The first two groups of interconnections are cycle connections which link processors within the same cycle. The third group is lateral connections which link processors between cycles. P[X, Y] is in dimension d if 2**(d-1) ≤ X < 2**d. By convention, if X=0, then P[0, Y] is in dimension 0. The total number of interconnections is at most (3/2)*N. An example of CCC(32) is given in Fig. 3.2.

3.3.1. Properties of the CCC

Consider the CCC as a graph in which the processors are the nodes, and the links are the edges. We establish two important properties of this graph.

Lemma 3.2: Every cube-connected cycles has a Hamiltonian path or cycle, and for all d, the dimensions of the first (2**d - 1)*2**r processors on the path are less than or equal to d.

Proof: We shall establish contructively that every CCC(N) has either an HP or an HC with the following properties:

(a) The HP or HC in CCC(N) includes the edge connecting P[X, 2**r - 1] and P[X, 0], for all X.

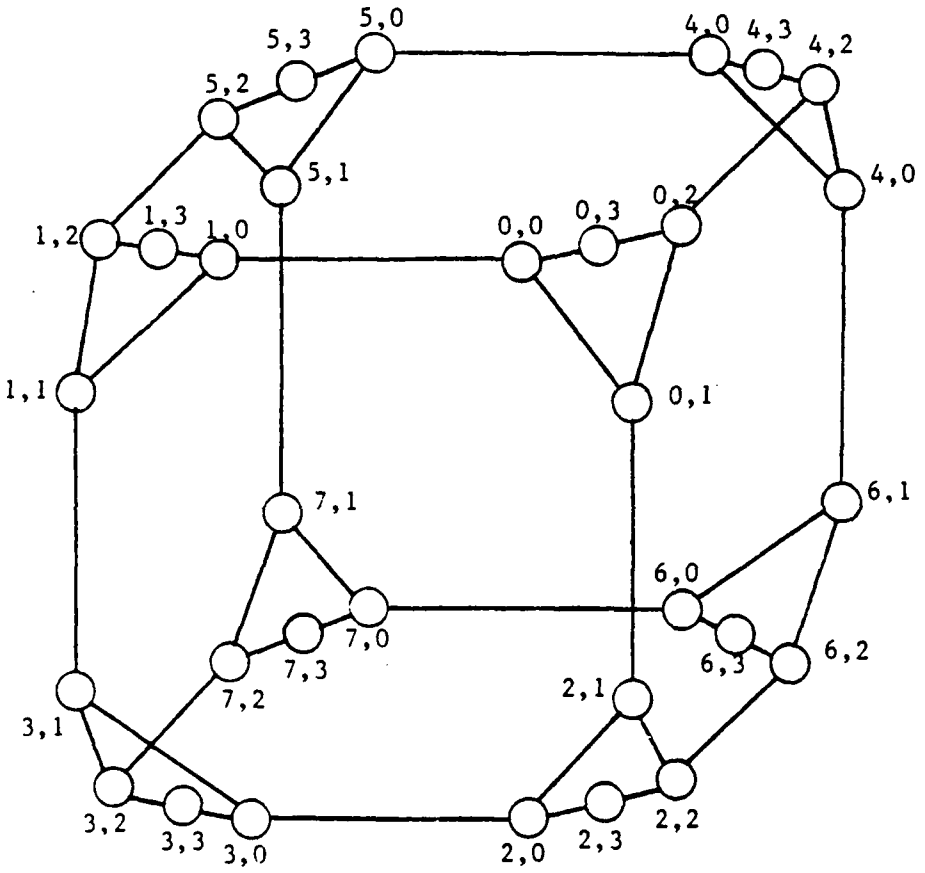


Fig. 3.2. A cube-connected cycles network with 32 processors. Processor Y in cycle X is denoted by X,Y.

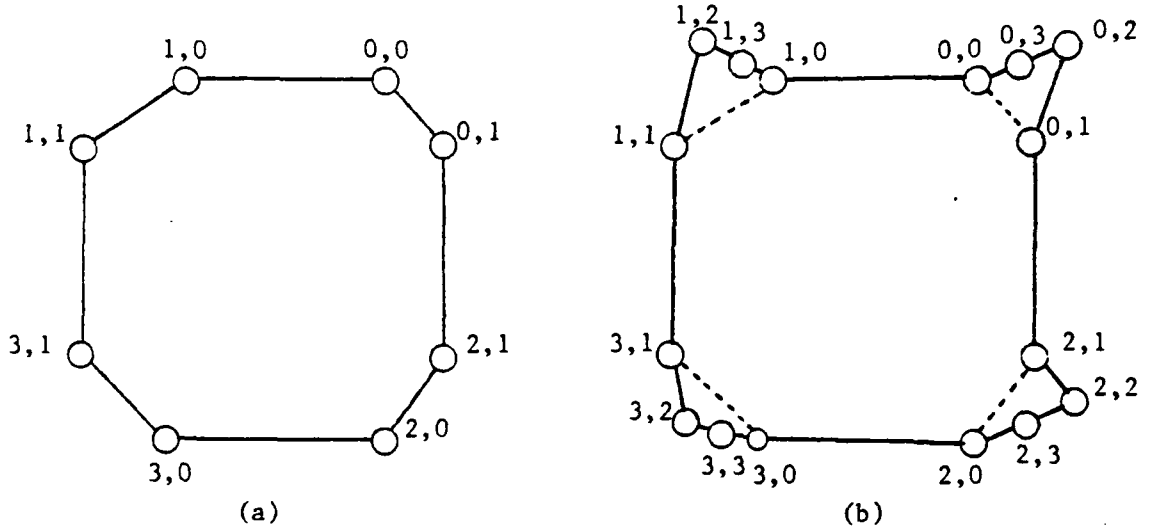
(b) The HP in CCC(N) begins at P[0, K-r] and ends at P[2**(K-r-1), K-r]. Furthermore, for all d, the dimensions of the first (2**d - 1)*2**r processors on the HP are less than or equal to d.

(c) The start node of the HC in CCC(N) can be chosen such that for all d, the first (2**d - 1)*2**r processors are less than or equal to d.

Evidently, CCC(4) has an HC that satisfies properties (a) and (b). Inductively assume that CCC(N/2) has either an HP or an HC satisfying property (a) and either (b) or (c). Consider the following two cases:

Case 1: r(N) = r(N/2) + 1. Let r = r(N). In this case CCC(N) is obtained from CCC(N/2) by doubling the length of each cycle. See Fig. 3.3(a) and 3.3(b) for an example. CCC(N/2) has an HC that satisfies properties (a) and (c), then construct an HC in CCC(N) by replacing, for every X, the path edge connecting P[X, 2**(r-1) - 1] and P[X, 0] in CCC(N/2) by the simple path P[X, 2**(r-1) - 1], P[X, 2**(r-1)] ,..., P[X, 2**r - 1], P[X, 0] in CCC(N). This path covers the 2**(r-1) new nodes introduced into the cycle. This HC in CCC(N) satisfies properties (a) and (c). Similarly, if CCC(N/2) has an HP that satisfies properties (a) and (b), by the same procedure we obtain an HP in CCC(N) that satisfies properties (a) and (b).

Case 2: r(N) = r(N/2). In this case CCC(N) is constructed from two



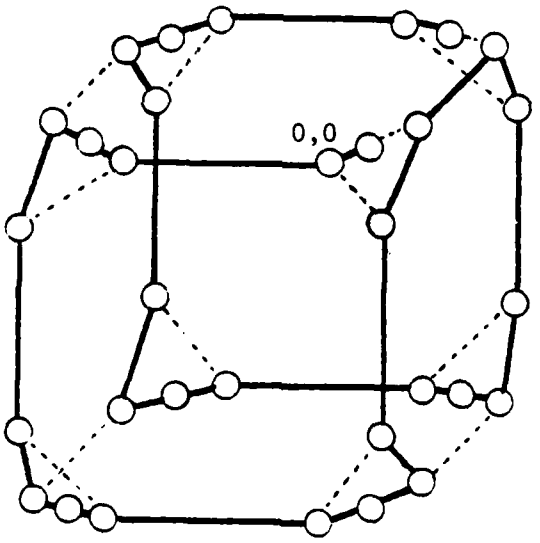


Fig. 3.3. (a) CCC(8). (b) CCC(16). (c) CCC(32). The solid line represents a Hamiltonian path or cycle.

copies of CCC(N/2) joined by lateral edges. P[X, Y] is in the first copy if X < 2**(K-r-1), in the second copy if X ≥ 2**(K-r-1).

First, suppose CCC(N/2) has an HC that satisfies properties (a) and (c). The HP in CCC(N) is obtained by taking the HC in each of the two copies of CCC(N/2) and

1. deleting from the HC in the first copy the path edge connecting P[0, K-r-1] to P[0, K-r].
2. deleting from the HC in the second copy the path edge connecting P[2**(K-r-1), K-r-1] to P[2**(K-r-1), K-r].
3. inserting a lateral path edge connecting P[0, K-r-1] in the first copy to P[2**(K-r-1), K-r-1] in the second copy.

This HP satisfies properties (a) and (b). See Fig. 3.3(b) and 3.3(c) for an example.

Second, suppose CCC(N/2) has an HP that satisfies properties (a) and (b). The HC in CCC(N) is obtained by taking the HP in each of the two copies of CCC(N/2) and

1. inserting a lateral path edge connecting P[0, K-r-1] in the first copy to P[2**(K-r-1), K-r-1] in the second copy.
2. inserting a lateral path edge connecting P[2**(K-r-2), K-r-1] in the first copy to P[3*2**(K-r-2), K-r-1] in the second copy.

This HC satisfies properties (a) and (c).

By the construction, notice that CCC(N) contains an HC if K-r is even, and an HP if K-r is odd. Also, case 1, CCC(N/2) an HP, need not occur.

Lemma 3.3: Every cube-connected cycles has no odd length circuits.

Proof: Every circuit in CCC(8) has even length. We proceed by induction on N. Assume inductively that CCC(N/2) has no odd length circuits.

Case 1: r(N) = r(N/2) + 1. Let r = r(N). In this case CCC(N) is obtained from CCC(N/2) by doubling the length of each cycle. The single edge connecting P[X,0] and P[X, 2**(r-1) - 1] in CCC(N/2), for every X, is replaced by a simple path PATH(X) connecting P[X, 2**(r-1) - 1], P[X, 2**(r-1)] ,..., P[X, 2**r - 1], P[X, 0] in CCC(N). If a circuit C in CCC(N) does not contain PATH(X) for every X, then C is the same as a circuit C' in CCC(N/2), and must be even length.

If a circuit does include PATH(X) for any X, then C does not contain the edge connecting P[X,0] to P[X,1]. Then C' can be constructed from C by replacing PATH(X) in CCC(N) by the single edge connecting P[X,0] and P[X, 2**(r-1) - 1] in CCC(N/2). Since PATH(X) is odd length, the length of C is greater than C' by an even number. If C' is odd length then C must also be odd length.

Case 2: r(N) = r(N/2). In this case CCC(N) is constructed from two copies of CCC(N/2), c1 and c2, joined by lateral edges. P[X, Y] is in c1 if X $\leq$ 2**(K-r-1), in c2 if X $\geq$ 2**(K-r-1). Suppose CCC(N) has a circuit C. C can induce a circuit C' in CCC(N/2) as follows:

1. Every edge of C that connects P[X, Y] to P[X', Y'], where Y = Y' = K-r-1, is ignored. These are lateral edges that join c1 and c2 in CCC(N). There must be an even number of them.
2. All other edges of C in CCC(N) are found in either c1 or c2. Each of these edges induce a corresponding edge in CCC(N/2).

Except for an even number of lateral edges, every edge of C is induced to an edge of C'. If C were odd length then C' would also be odd length. This contradicts the inductive hypothesis that C is even length.

3.4. Dictionary Machine on the CCC

The dictionary algorithm of Section 3.1 is applied to the cube-connected cycles network as follows: the data path P(0), P(1),..., P(N-1) coincides with the Hamiltonian path or cycle of the CCC established by lemma 3.2. Let r be the smallest integer such that r**2 + r $\geq$ K, where K = log N. The CCC contains an HC if K-r is even, an HP if K-r is odd. If K-r is odd then P(0) is identified with P[0, K-r] ,..., P(N-1) with P[2**(K-r-1), K-r]. If K-r is even then P(0) is identified with P[0,0] ,..., P(N-1) with P[0,2**r - 1]. Incoming and outgoing ports are computed during the initialization step.

See Section 3.2.2. Lemma 3.3 ensures that instructions and responses are pipelined through the CCC, free of contention or timing problems. Every processor contains a buffer and uses retiming (see Section 3.2.3) to synchronize the execution of instructions. Fig 3.4 shows the data path and forward links for CCC(32).

3.4.1. Latency Analysis

Two important performance criteria are used to evaluate a dictionary machine: the pipeline interval and the latency. In a sequence of instructions I1,I2,I3,... the pipeline interval is the issue time between two successive instructions. The dictionary algorithm can process an instruction in four steps, so the pipeline interval is also a constant number of steps, independent of the size of the dictionary machine. The latency is the time between an instruction being issued and its reponse being output from the machine—the "end-to-end" time of the pipeline.

The latency depends on the number of records currently stored in the dictionary, n, and is proportional to the maximum distance d(max) from root to all processors currently holding either a record or a hole. The farthest processor, P(max), will respond to a query in 2*d(max) time.

We determine the latency for a CCC(N) when n records are currently stored in the dictionary.

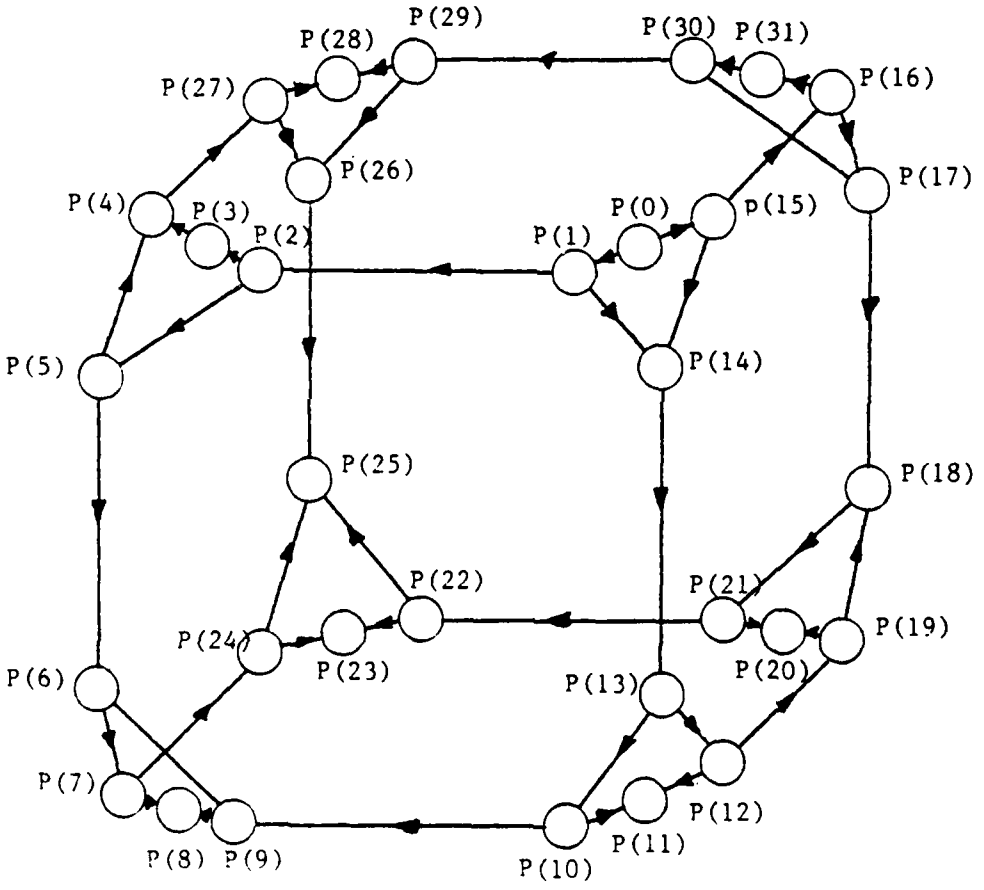


Fig. 3.4. A CCC(32). The arrows point in the direction of the forward links. P(0), P(1),..., P(31) is the data path.

Let r be the smallest integer such that $2^{**}r + r \geq \log N$.

Let d be the smallest integer such that $(2^{**}d - 1)*2^{**}r \geq 2*n$.

It follows that $d \geq 1 + \log(n/(2^{**}r))$.

Since $r = O[\log N]$ then $d = O[\log(n/\log N)]$.

By lemma 3.2, the dimensions of the first $(2^{**}d - 1)*2^{**}r$ processors on the path are less than or equal to d. This implies that every processor holding a record is in the first d dimensions of CCC(N). So the distance from P(0) to P(max) is bounded by d times the length of each cycle. Thus;

$$\text{Latency} = d*2^{**}r = O[(\log(n/\log N))*(\log N)].$$

3.4.2. Returning Responses

Suppose the responses were sent directly to P(0). If instruction Ii were a membership query whose response Ri resided in P(max), and Ii+1 were an extract operation whose response Ri+1 resided in P(1), then Ri+1 would be returned before Ri. The responses would be returned in the wrong order.

To ensure that responses are returned in chronological order, they are first forwarded to the "bottom" of the network, then returned along the back links. The bottom is the group of processors whose distance from P(0) is d(max). This procedure guarantees that the round trip distance of every query/response is equal to 2*d(max).

To find the bottom of the CCC, P(0) estimates the value of d(max). At each step, concurrent with the execution of instructions and the delivery of reponses, every processor P delivers to its predecessor(s) the maximum value of its own distance, d(P), and the distances received by its successors in the previous step. d(P) is masked to zero if P contains no record. In this manner, P(0) receives at each step a value D which represents the "current" d(max) value as it was 2*D steps ago. Since an instruction is processed in four steps, the number of instructions that were issued during these 2*D steps is (1/2)*D. At worst, the current d(max) can be (3/2)*D. So we estimate d(max) to be (3/2)*D.

We use this estimate to return responses. P(0) attaches a counter to each instruction. Its value is initially set to (3/2)*D. The counter specifies how far a response for the instruction must travel until it returns to P(0). Thus, when a response is generated it proceeds deeper into the machine just as if it were an instruction. The counter decreases by one whenever the instruction (or response that replaces the instruction) is forwarded. When the counter becomes zero, the response is returned along the back links to P(0), which emits the response. Every response travels (3/2)*D to its furthest point. This ensures that an instruction will reach every processor that currently holds a record, even if the previous (1/2)*D instructions are nonredundant insertions.

3.5. Remark

The latency of the CCC could be reduced by introducing an extra link into each cycle. The tradeoff is to increase the VLSI layout area of the CCC.

CHAPTER 4

CONCLUSION

4.1. Summary

Table 4.1 shows the latencies and VLSI layout areas for our two cube-class dictionary machines. Shown for comparison are several tree machines and a mesh network. N is the maximum capacity of the machine, while n is the number of keys stored in the dictionary at a particular time.

TABLE 4.1. Latencies and areas of dictionary machines.

Network	Latency	Area
Cube-connected cycles	$O[(\log(n/\log N))*\log N]$	$O[N^2/\log^2 N]$
Shuffle-exchange network	$O[\log N]$	$O[N^2/\log^2 N]$
Systolic array-tree (Leiserson, 1979)	$O[\log N]$	$O[N]$
X-tree (Ottmann et al., 1982)	$O[\log n]$	$O[N]$
Tree (Atallah & Kosaraju, 1985)	$O[\log n]$	$O[N]$
Mesh	$O[n^{1/2}]$	$O[N^{1/2}]$

The mesh has a low chip area but takes a long time to return answers. The SEN and CCC machines have larger latencies and areas than the tree machines of Ottmann et al. (1982) and Atallah and Kosaraju (1985). Nevertheless, the algorithms presented in this thesis are useful when only the shuffle-exchange network or the cube-connected cycles are available for general purpose computing. Furthermore, the novel pipelined architecture of Chapter 3 may be applicable in other situations.

4.2. Further Research

The orthogonal trees network (OTN) (Leighton, 1981; Nath, Maheshwari & Bhatt, 1983) can efficiently solve a large class of problems such as sorting, matrix multiplication, FFT, and finding connected components in a graph, etc. In some problems the OTN outperforms both the SEN and the CCC. It is also easy to program and amenable to pipelining.

The OTN consists of an NxN matrix of processors in which each row and each column of processors form the leaves of a binary tree. Because the OTN is a generalization of the tree network, it can emulate all the tree machines previously described. The efficiency of the OTN for maintaining dictionary operations is still undetermined. For instance, can the 2*N trees of the OTN share a single database or must the database be limited to a single tree?

REFERENCES

Atallah, M.J., and Kosaraju, S.R. (1985), "A generalized dictionary machine for VLSI," IEEE Trans. Comput., vol. c-34, pp. 151-155.

Batcher, K.E. (1968), "Sorting networks and their applications," Proc. AFIPS Spring Joint Computer Conference, vol. 32, AFIPS Press, Montvale, NJ, pp. 307-314.

Carey, M.J., and Thompson, C.D. (1984), "An efficient implementation of search trees on [lg N + 1] processors," IEEE Trans. Comput., vol. c-33, pp. 1038-1041.

Fisher, A.L. (1984), "Dictionary machines with a small number of processors," Proc. 11th Ann. IEEE Symp. on Computer Architecture, Ann Arbor, MI, pp. 151-156.

Kleitman, D., Leighton, F.T., Lepley, M., and Miller, G.L. (1983), "An asymptotically optimal layout for the shuffle-exchange graph," J. Comput. Syst. Sci., vol. 26, pp. 339-361.

Knuth, D.E. (1973), The Art of Computer Programming, vol. 3, Sorting and Searching, Addison-Wesley, Reading, MA.

Kung, H.T. (1982), "Why systolic architectures?" IEEE Computer. vol. 15, pp. 37-48.

Leighton, F.T. (1981) "New lower bound techniques for VLSI," Proc. 22nd Ann. IEEE Symp. Foundations Comput. Sci., pp. 1-12.

Leiserson, C.E. (1979), "Systolic priority queues," Tech. Rep. CMU-CS-79-115, Dept. Computer Science, Carnegie-Mellon Univ.

Leiserson, C.E., and Saxe, J.B. (1984), "Optimizing synchronous sytems," J. VLSI Comput. Syst., vol. 1, pp. 41-67.

Nath, D., Maheshwari, S.N., and Bhatt, P.C.P. (1983), "Efficient VLSI networks for parallel processing based on orthogonal trees," IEEE Trans. Comput., vol. c-32, pp. 569-581.

Ottmann, T.A., Rosenberg, A.L., and Stockmeyer, L.J. (1982), "A dictionary machine (for VLSI)," IEEE Trans. Comput., vol. c-31, pp. 892-897.

Preparata, F.P., and Vuillemin, J. (1981), "The cube-connected cycles: A versatile network for parallel computation," Commun. ACM, vol. 25, pp. 300-309.

Schwartz, J.T. (1980), "Ultracomputers," ACM Trans. Program. Lang. Syst., vol. 2, pp. 484-521.

Somani, A.K., and Agarwal, V.K. (1984), "An efficient VLSI dictionary machine," Proc. 11th Ann. IEEE Symp. on Computer Architecture, Ann Arbor, MI, pp. 142-150.

Stone, H.S. (1971), "Parallel processing with the perfect shuffle," IEEE Trans. Comput., vol. c-20, pp. 153-161.

Synder, L. (1982), "Introduction to the configurable highly parallel computer," IEEE Computer, vol. 15, pp. 47-56.

END

FILMED

1-86

DTIC